### LAMBDA FUNCTION -> **SuffledQuestionUser**

API URL -> <https://ybkfar4y6i.execute-api.us-east-1.amazonaws.com/studentq/SuffledQuestionUser>

Resource -> **StudentAPI -> SuffledQuestionUser**

**Stage ->** student

**Role of the API**:

* This API endpoint fetches a quiz by ID from MongoDB, shuffles its MCQ and descriptive questions using the Fisher-Yates algorithm, removes correct answers, and then updates a student's document with the shuffled questions.

**Functioning**:

* Validates the JWT token provided in the request headers for authorization.
* Connects to MongoDB using the URI provided in the db environment variable.
* Validates the request payload to ensure it contains necessary fields (id and applicationNumber).
* Retrieves the quiz from MongoDB using Quiz.findById.
* Shuffles both MCQs and descriptive questions retrieved from the quiz using the Fisher-Yates shuffle algorithm.
* Removes correct answers from the shuffled MCQs and removes answers from descriptive questions for security reasons.
* Constructs a response object containing the shuffled questions and quiz duration.
* Updates the Student document with the shuffled questions and quiz details.
* Returns the shuffled questions and quiz duration as a response to the client.

**Request Body**:

* The API expects a JSON object in the request body with the following fields:
  + id: ID of the quiz to fetch from MongoDB.
  + applicationNumber: Application number of the student to update the Student document.

**Response**:

* **200 OK**: If the quiz is successfully retrieved and shuffled, returns the shuffled questions and quiz duration.
* **400 Bad Request**: If the request payload is missing required fields.
* **401 Unauthorized**: If the JWT token is missing or invalid.
* **404 Not Found**: If the quiz with the provided ID or student with the provided application number is not found.
* **500 Internal Server Error**: If there's any server-side issue during quiz retrieval, shuffling, or database operation.

**Logic**:

* Handles CORS preflight requests (OPTIONS method) to allow CORS headers.
* Validates the JWT token for authorization.
* Connects to MongoDB using Mongoose for database operations.
* Validates the incoming payload to ensure all required fields are present.
* Retrieves the quiz from MongoDB using Quiz.findById.
* Shuffles MCQs and descriptive questions using the Fisher-Yates algorithm.
* Removes correct answers from MCQs and answers from descriptive questions for security.
* Updates the Student document with the shuffled questions and quiz details.
* Returns appropriate status codes and messages based on the success or failure of these operations.

**Dependencies**:

* mongoose: MongoDB object modeling tool for Node.js.
* jsonwebtoken: For generating and verifying JSON Web Tokens (JWT).
* Quiz: Mongoose model representing the Quiz collection in MongoDB.
* Student: Mongoose model representing the Student collection in MongoDB.

CODE ->

const mongoose = require('mongoose');

const jwt = require('jsonwebtoken');

const Quiz = require('question'); // Adjust the path if necessary

const Student = require('Student'); // Adjust the path if necessary

// MongoDB connection management

let cachedDb = null;

const connectToDatabase = async () => {

if (cachedDb) {

return cachedDb;

}

try {

const dbUri = process.env.db; // Replace with your MongoDB URI

cachedDb = await mongoose.connect(dbUri);

return cachedDb;

} catch (error) {

console.error('Error connecting to MongoDB:', error);

throw new Error('Could not connect to MongoDB');

}

};

// Fisher-Yates Shuffle Algorithm

function shuffleArray(array) {

const shuffledArray = array.slice(); // Create a copy of the array

const currentTime = new Date().getTime(); // Get the current time in milliseconds

for (let i = shuffledArray.length - 1; i > 0; i--) {

const randomTimeFactor = Math.floor(currentTime \* Math.random());

const j = Math.floor(randomTimeFactor % (i + 1));

[shuffledArray[i], shuffledArray[j]] = [shuffledArray[j], shuffledArray[i]];

}

return shuffledArray;

}

// Remove answers from descriptive questions

function removeDescriptiveAnswers(descriptiveQuizz) {

return descriptiveQuizz.map(descriptive => {

const { answer, ...rest } = descriptive.toObject();

return rest;

});

}

// Remove correct answers from MCQ questions

function removeCorrectAnswers(mcqQuizz) {

return mcqQuizz.map(mcq => {

const { correctAnswer, ...rest } = mcq.toObject();

return rest;

});

}

// Validate request payload

function validatePayload(payload) {

if (!payload.applicationNumber) {

throw new Error('Application number is required');

}

if (!payload.id) {

throw new Error('Quiz ID is required');

}

}

exports.handler = async (event) => {

try {

// Handle preflight CORS request

if (event.httpMethod === 'OPTIONS') {

return {

statusCode: 204,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins, update as necessary

'Access-Control-Allow-Methods': 'POST, OPTIONS', // Allow specific methods

'Access-Control-Allow-Headers': 'Content-Type, Authorization', // Allow specific headers

},

body: null,

};

}

// Validate JWT token

const token = event.headers.Authorization || event.headers.authorization;

if (!token) {

return {

statusCode: 401,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: 'Authorization token missing' }),

};

}

let decoded;

try {

decoded = jwt.verify(token, process.env.JWT\_SECRET\_KEY); // Replace with your JWT secret

} catch (err) {

return {

statusCode: 401,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: 'Invalid token' }),

};

}

// Connect to MongoDB

await connectToDatabase();

// Parse and validate the request body

const requestBody = JSON.parse(event.body);

try {

validatePayload(requestBody);

} catch (validationError) {

return {

statusCode: 400,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: validationError.message }),

};

}

const { id, applicationNumber } = requestBody;

// Fetch quiz by ID

const quiz = await Quiz.findById(id, '\_id mcqQuizz descriptiveQuizz quizDuration').exec();

if (!quiz) {

return {

statusCode: 404,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: 'Quiz not found' }),

};

}

// Shuffle the MCQ and descriptive questions

const shuffledMCQs = shuffleArray(quiz.mcqQuizz);

const shuffledDescriptive = shuffleArray(quiz.descriptiveQuizz);

// Remove correct answers from shuffled MCQs for response

const mcqQuizzWithoutAnswers = removeCorrectAnswers(shuffledMCQs);

// Remove answers from shuffled descriptive questions for response

const descriptiveQuizzWithoutAnswers = removeDescriptiveAnswers(shuffledDescriptive);

// Construct response

const response = {

\_id: quiz.\_id,

mcqQuizz: mcqQuizzWithoutAnswers,

descriptiveQuizz: descriptiveQuizzWithoutAnswers,

quizDuration: quiz.quizDuration

};

// Find the student document with matching applicationNumber and update it

const studentdetails = await Student.findOneAndUpdate(

{ applicationNumber: applicationNumber },

{ $set: { generatedShuffledQuestion: { ...response, mcqQuizz: shuffledMCQs, descriptiveQuizz: shuffledDescriptive } } },

{ new: true }

);

if (!studentdetails) {

return {

statusCode: 404,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: 'Student with given application number not found' }),

};

}

return {

statusCode: 200,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify(response),

};

} catch (error) {

console.error('Error retrieving quiz:', error);

return {

statusCode: 500,

headers: {

'Access-Control-Allow-Origin': '\*', // Allow all origins

},

body: JSON.stringify({ error: 'Internal Server Error' }),

};

}

};

MODEL -> 1 -> question.js

const mongoose = require('mongoose');

// Define schema for multiple choice questions

const MCQSchema = new mongoose.Schema({

question: {

type: String,

required: false,

},

questionImageLink: {

type: String,

required: false,

},

options: [{

answer: {

type: String,

required: false,

},

answerImageLink: {

type: String,

required: false,

}

}],

correctAnswer: {

type: String,

required: true,

},

description: {

type: String,

required: false

},

version: {

type: Number,

default: 1

}

});

// Define schema for descriptive questions

const DescriptiveSchema = new mongoose.Schema({

question: {

type: String,

required: false,

},

questionImageLink: {

type: String,

required: false,

},

answer: {

type: String,

required: false

},

answerImageLink: {

type: String,

required: false,

},

version: {

type: Number,

default: 1

}

});

// Define main quiz schema

const QuizSchema = new mongoose.Schema({

quizTitle: {

type: String,

required: true,

},

creatorName: {

type: String,

required: false

},

creatorEmail: {

type: String,

required: true

},

isCompleted: {

type: Boolean,

required: false,

default: false

},

status: {

type: Boolean,

required: false,

default: true

},

preVersionID: {

type: [mongoose.Schema.Types.ObjectId],

default: []

},

mcqQuizz: [MCQSchema], // Array of multiple choice questions

descriptiveQuizz: [DescriptiveSchema], // Array of descriptive questions

quizDuration: {

type: Number,

required: false

}

}, { timestamps: { createdAt: 'createdAt' } });

// Pre-save middleware to set creatorName from User2's fullname

QuizSchema.pre('save', async function(next) {

if (this.isNew || this.isModified('creatorEmail')) {

const User2 = mongoose.model('User2');

const user = await User2.findOne({ email: this.creatorEmail }).exec();

if (user) {

this.creatorName = user.fullname;

} else {

const error = new Error('User not found');

error.statusCode = 404;

return next(error);

}

}

next();

});

// Middleware to handle versioning of questions in mcqQuizz and descriptiveQuizz

QuizSchema.pre('save', async function(next) {

if (!this.isNew) {

const originalQuiz = await mongoose.model('Quiz').findById(this.\_id).exec();

if (originalQuiz) {

this.mcqQuizz.forEach(mcq => {

const originalMCQ = originalQuiz.mcqQuizz.id(mcq.\_id);

if (originalMCQ && !mcq.\_id.equals(originalMCQ.\_id)) {

mcq.version = originalMCQ.version + 1;

}

});

this.descriptiveQuizz.forEach(dq => {

const originalDQ = originalQuiz.descriptiveQuizz.id(dq.\_id);

if (originalDQ && !dq.\_id.equals(originalDQ.\_id)) {

dq.version = originalDQ.version + 1;

}

});

// Handle deleted questions by adding their IDs to preVersionID

originalQuiz.mcqQuizz.forEach(originalMCQ => {

if (!this.mcqQuizz.id(originalMCQ.\_id)) {

this.preVersionID.push(originalMCQ.\_id);

}

});

originalQuiz.descriptiveQuizz.forEach(originalDQ => {

if (!this.descriptiveQuizz.id(originalDQ.\_id)) {

this.preVersionID.push(originalDQ.\_id);

}

});

}

}

next();

});

// Create a model using the schema

const Quiz = mongoose.model('Quiz', QuizSchema);

module.exports = Quiz;

MODEL -> Student.js

const mongoose = require('mongoose');

const studentSchema = new mongoose.Schema({

fullName: {

type: String,

required: true,

trim: true

},

applicationNumber: {

type: String,

required: true,

unique: true,

trim: true

},

DOB: {

type: Date,

required: true

},

questionId: {

type: String,

required: true,

trim: true

},

generatedShuffledQuestion: {

type: mongoose.Schema.Types.Mixed, // Using Mixed type to allow flexibility in the structure

required: false

}

}, {

timestamps: true

});

// Setting applicationNumber as the primary key

studentSchema.index({ applicationNumber: 1 }, { unique: true });

const Student = mongoose.model('StudentDetails', studentSchema);

module.exports = Student;